HIGH ORDER SEISMIC SIMULATIONS ON THE INTEL® XEON PHI™ PROCESSOR (KNIGHTS LANDING) & EFFICIENCY OF HIGH ORDER SPECTRAL ELEMENT METHODS ON PETASCALE ARCHITECTURES

Alexander Heinecke, Alexander Breuer, Michael Bader, Pradeep Dubey
&
Maxwell Hutchinson, ***Alexander Heinecke***, Hans Pabst, Greg Henry, Matteo Parsani, and David Keyes

Parallel Computing Lab, Intel Labs, USA

Intel® Xeon Phi™ Processor

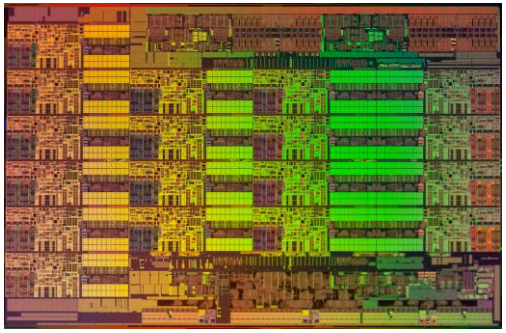
(code-named Knights Landing)

Current & Next Generation Intel® Xeon and Xeon Phi™ Platforms

Xeon*

Latest released – Broadwell (14nm process)

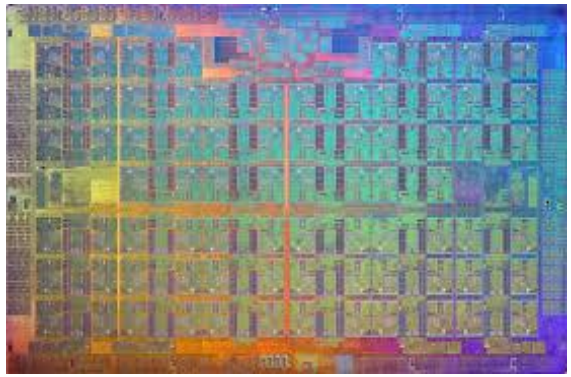
- Intel's Foundation of HPC Performance
- Up to 22 cores, Hyperthreading
- ~66 GB/s stream memory BW (4 ch. DDR4 2400)
- AVX2 – 256-bit (4 DP, 8 SP flops) -> >0.7 TFLOPS
- 20 PCIe lanes



Xeon Phi*

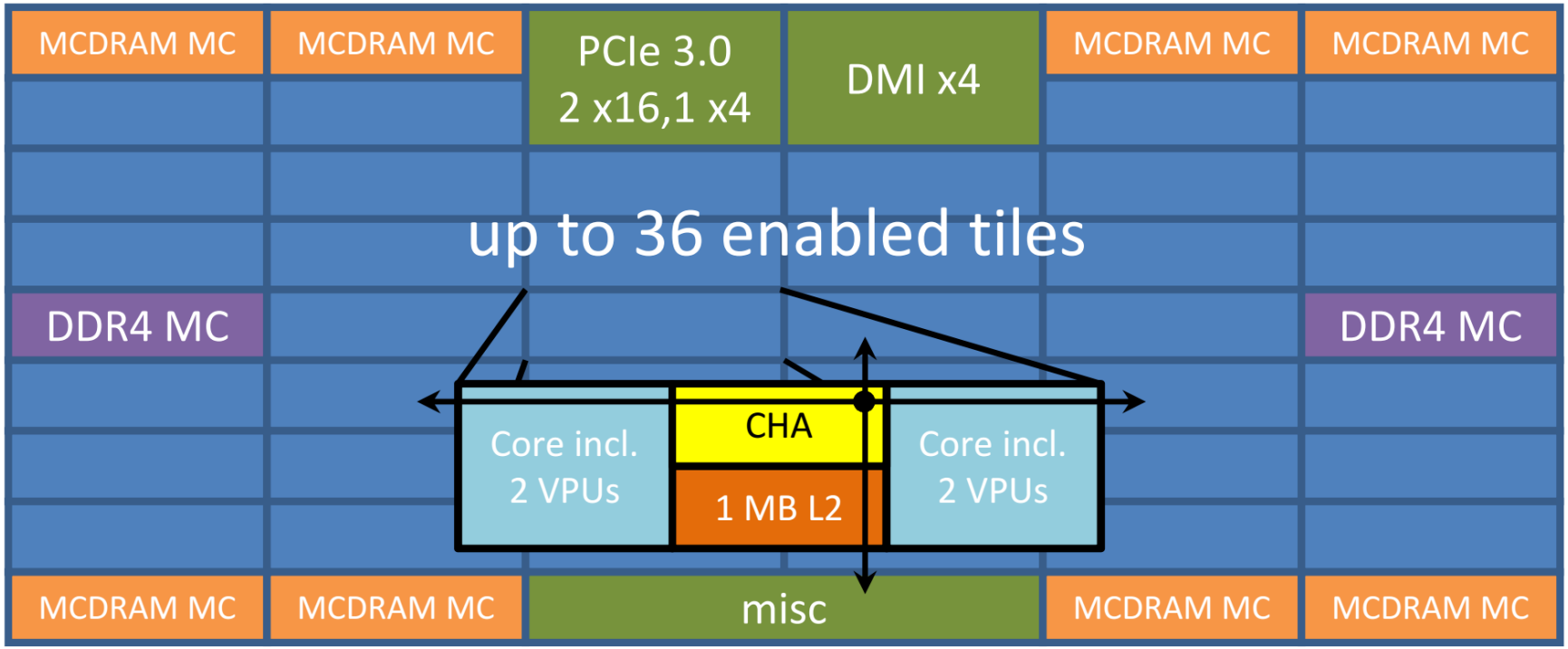
Knights Landing (14nm process)

- Optimized for highly parallelized compute intensive workloads
- Common programming model & S/W tools with Xeon processors, enabling efficient app readiness and performance tuning
- up to 72 cores, 490 GB/s stream BW, on-die 2D mesh
- AVX512– 512-bit (8 DP, 16 SP flops) -> >3 TFLOPS
- 36 PCIe lanes



*Intel Xeon and Intel Xeon Phi are trademarks of Intel Coporation in the US and/or other countries.

Intel Xeon Phi processor (Knights Landing)



- Self boot and binary compatible with main line IA (x87, SSE, AVX, AVX512)
- Boots standard OS, such as Linux or Windows.
- Significant improvement in scalar (heavily-modified Intel Atom core) and vector (2x AVX512 VPU per core) performance.
- Memory on package: innovative memory architecture for high bandwidth and high capacity
- Fabric on package connected with 32 PCIe Gen 3 lanes for 200 Gbps

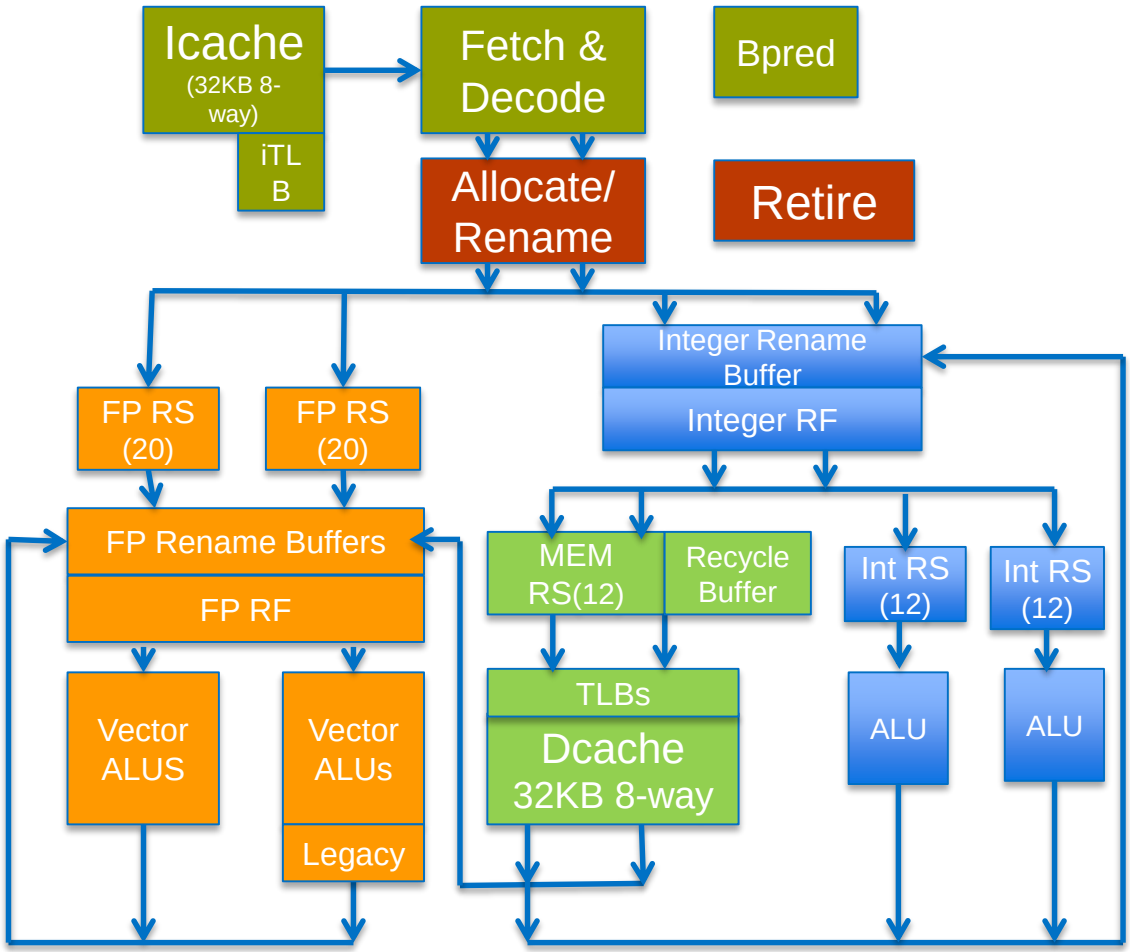
Core & VPU

Balanced power efficiency, single thread performance and parallel performance

2-wide Out-of-order core

4 SMT threads

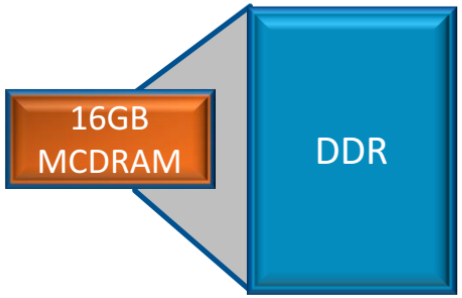
- 72 in-flight instructions.
- 6-wide execution
- 64 SP and 32 DP Flop/cycle
- Dual ported DL1 → to feed 2 VPU
- Two-level TLB. Large page support
- Gather/Scatter engine
- Unaligned load/store support
- Core resources **shared** or **dynamically repartitioned** between active threads
- General purpose IA core



Memory Modes of Xeon Phi 72xx

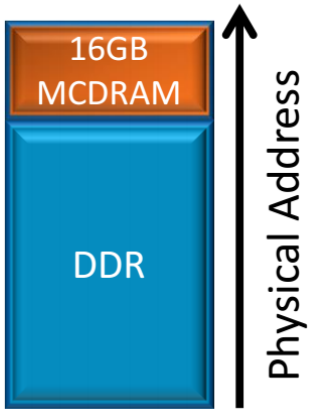
Three Modes. Selected at boot

Cache Mode



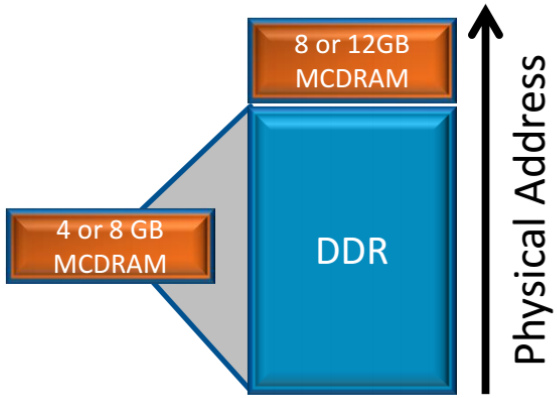
- SW-Transparent, Mem-side cache
- Direct mapped. 64B lines.
- Tags part of line
- Covers whole DDR range

Flat Mode



- MCDRAM as regular memory
- SW-Managed
- Same address space

Hybrid Mode



- Part cache, Part memory
- 25% or 50% cache
- Benefits of both

Taken from: A. Sodani: Knights Landing (KNL): 2nd Generation Intel Xeon Phi Processor, Hotchips '15

SeisSol and LIBXSMM

SeisSol's Compute Kernels – Time Integration

Recursive Scheme

$$\mathcal{T}_k := \mathcal{T}_k\left(Q_k^n, \Delta t\right) = \sum_{j=0}^{\mathcal{O}-1} \frac{\Delta t^{j+1}}{(j+1)!} \frac{\partial^j}{\partial t^j} Q_k(t^n)$$

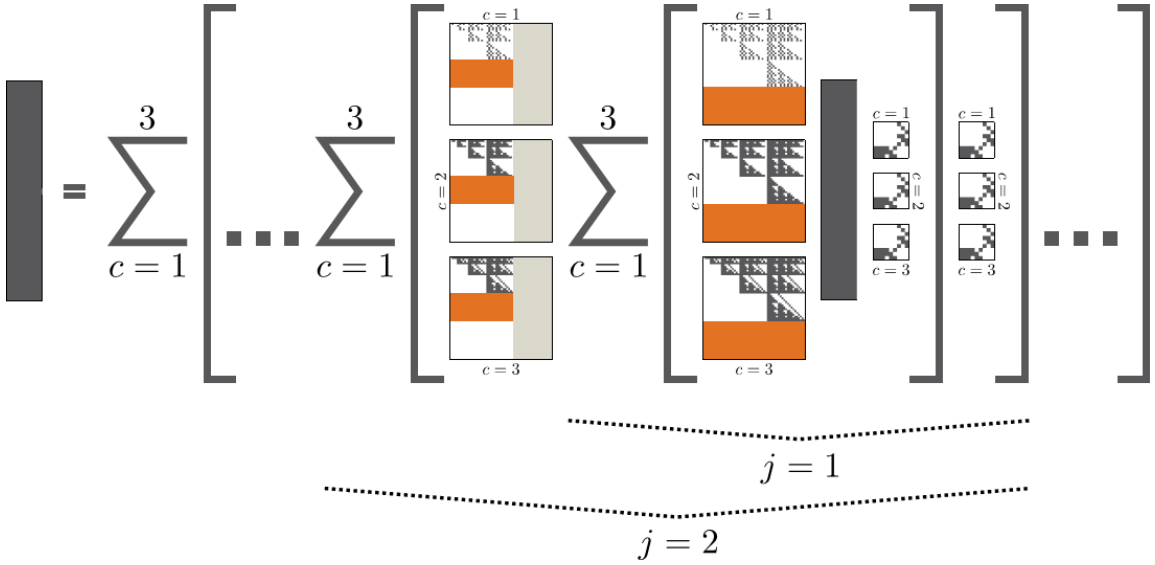
$$\frac{\partial^{j+1}}{\partial t^{j+1}} Q_k = -\sum_{c=1}^{3} \hat{K}^{\xi_c} \left(\frac{\partial^j}{\partial t^j} Q_k\right) A^{\star}_{k,c}. \quad \text{with} \quad \frac{\partial^0}{\partial t^0} Q_k = Q_k^n$$

Zero blocks in $\widehat{K}^{\xi_1}$, $\widehat{K}^{\xi_2}$ and $\widehat{K}^{\xi_3}$ lead to zero values in the degrees of freedom Q_k^n

Matrix size is reduced in each recursive step

Zero values in Q_k^n also appear in the multiplications with $A^*_{k,1}$, $A^*_{k,2}$, and $A^*_{k,3}$

Typical Matrix sizes of production runs (converge

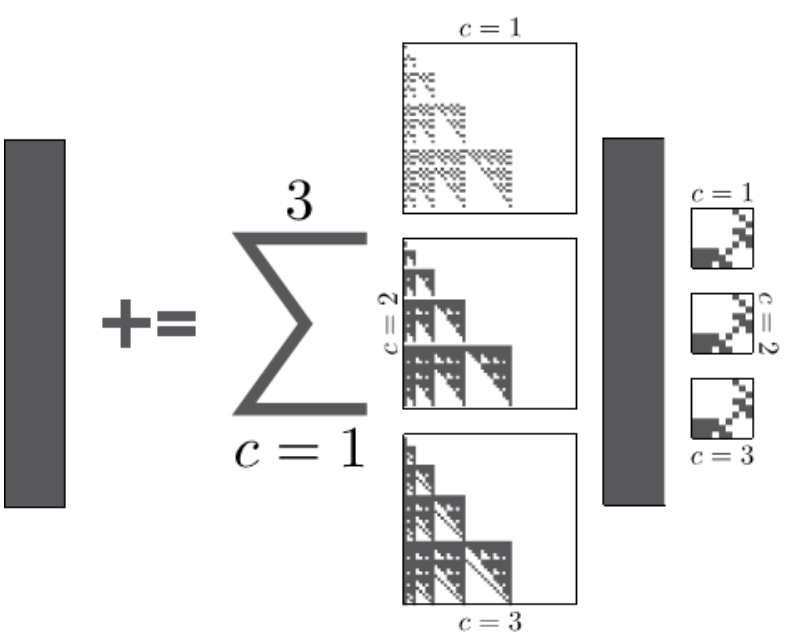


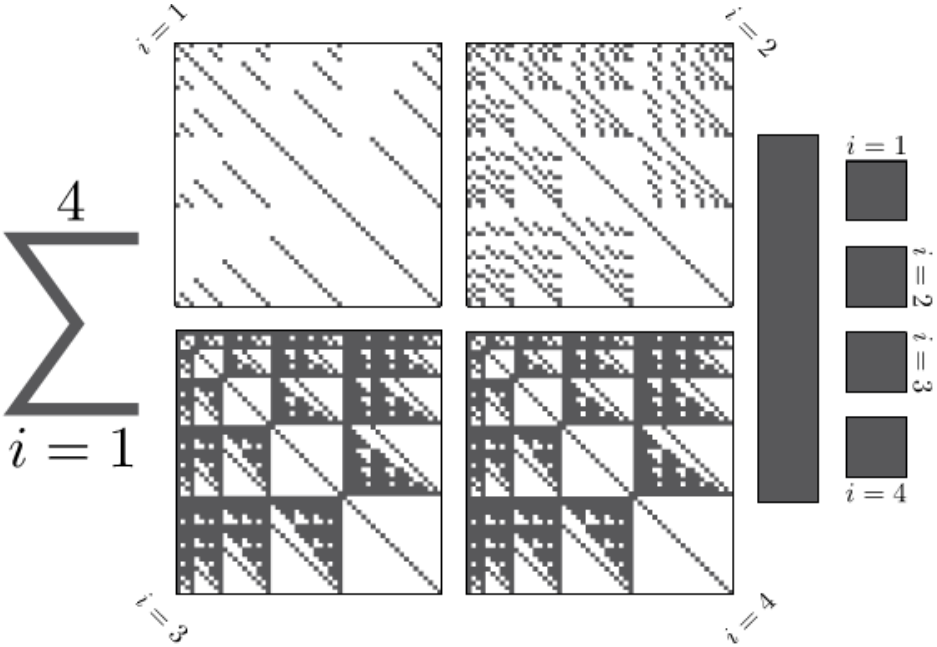
SeisSol's Compute Kernels – Local Integration

Small Matrix-Matrix multiplications

Typical Matrix sizes of production runs (convergence order 6): $9x9, 56x9, 56x35$

A priori known sparsity patterns



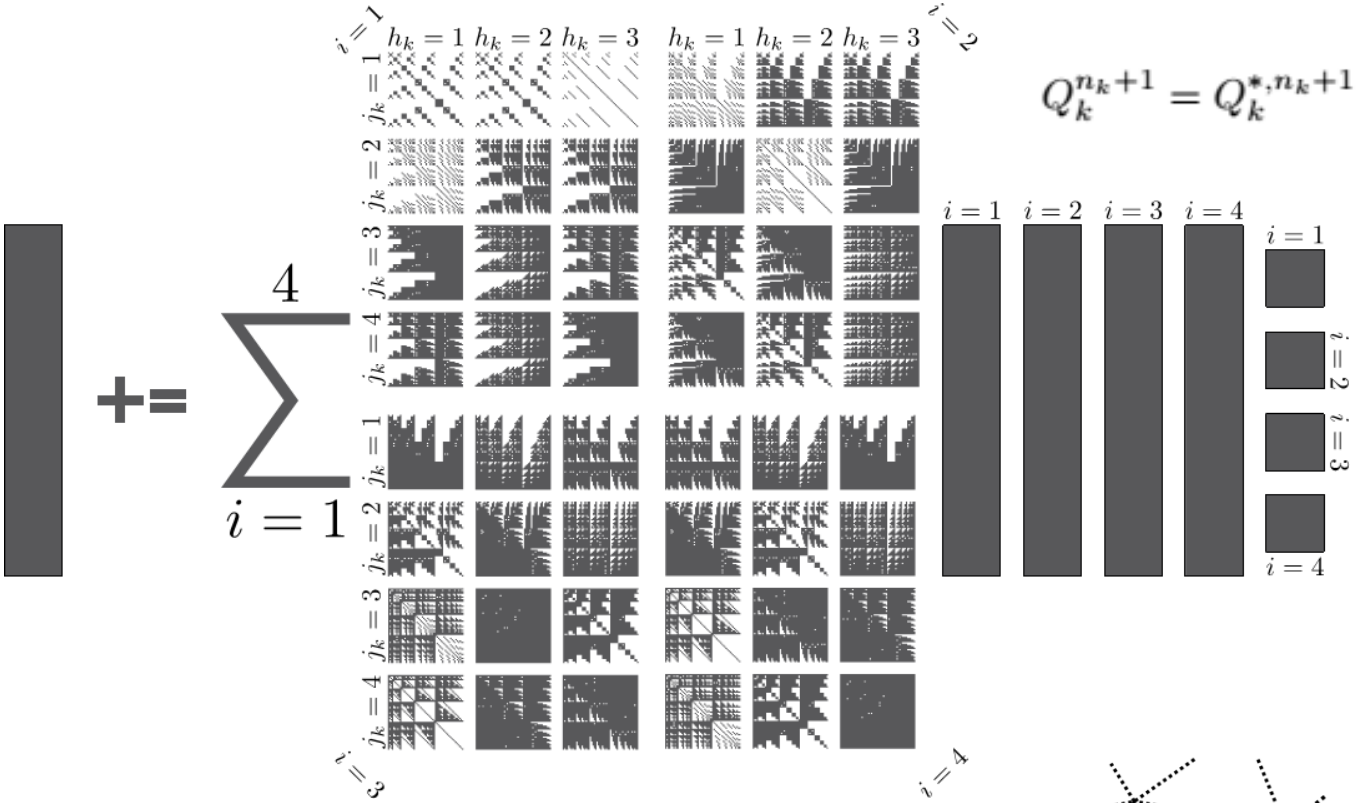


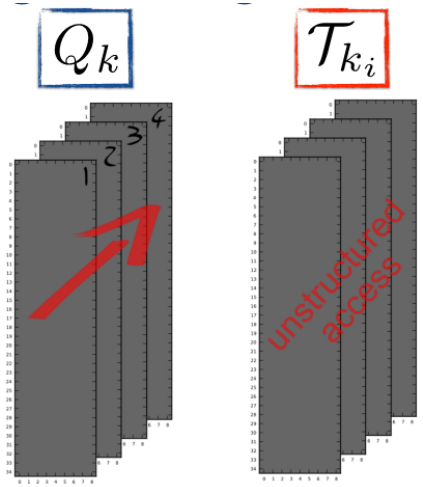
$$\mathcal{V}_k(\mathcal{T}_k) = \sum_{c=1}^{3} \tilde{K}^{\xi_c}(\mathcal{T}_k) A_k^{\xi_c}$$

$$Q_k^{*,n_k+1} = Q_k^{n_k} + \mathcal{V}_k - \sum_{i=1}^{4} \hat{F}^{-,i}(\mathcal{T}_k) \hat{A}_{k,i}^{-}$$

SeisSol's Compute Kernels – Neighbor Integration

$$Q_k^{n_k+1} = Q_k^{*,n_k+1} - \sum_{i=1}^{4} \hat{F}^{+,i,j_k,h_k}\left(\mathcal{T}_{k_i}\right) \hat{A}_{k,i}^{+}$$





Unstructured Access to:

- Flux matrices
- Time integrated DOFs of neighboring elements

SeisSol Key Compute Kernels

- **Local Integration** (partly L1-cache BW bound, linear memory accesses)
  - Consist of time integration, volume integration and local part of boundary integration
  - 9 element local matrices (3 9x9, 4 9x9, 2 56x9) and 10 global matrices (3 40x56, 3 35x56, 4 56x56) (all number for sixth order)
  - **Flop/byte approx. 15** (sixth order)
- **Neighbor Integration** (in theory compute bound, irregular memory accesses and higher bandwidth)
  - 10 element local matrices (4 9x9, 6 56x9) and 4 out of 48 global matrices (4 56x56)
  - **Flop/byte approx. 2** (sixth order) on Xeon Phi as we cannot keep the 48 matrices in L2
- Sparse and Dense Matrix Multiplication of small sizes:
  - Due to unstructured meshes we need a prefetching stream that matches the mesh structure (not a regular DGEMM prefetching strategy)
  - Global (irregularly accessed) operators occupy close to 1.5 MB

  → **Intel MKL cannot by used, due to blocking and padding overheads. Using MKL instead of our highly tuned kernels results into 1.5-3X speed-down depending on order.**

  → **Batched GEMM is not beneficial as it would mean loosing locality**

  → **We leverage and optimized LIBXSMM to speed-up SeisSol's compute kernels**

Simultaneous Use of DDR4 and MCDRAM

- Low bandwidth requirements in local integration, high bandwidth requirements in neigh integration (behavior changes with convergence order!)
- Idea: Place as few as possible data structures in MCDRAM to make efficient use of MCDRAM, e.g. running large problem exceeding MCDRAM, we use memkind for this, https://github.com/memkind/memkind

order	Q_k	$\mathcal{B}_k, \mathcal{D}_k$	$A_k^{\xi_c}, \hat{A}_k^{-,i}, \hat{A}_k^{+,i}$	$\hat{K}^{\xi_c}, \tilde{K}^{\xi_c}, \hat{F}^{-,i}, \hat{F}^{+,i,j,h}$
2	MCDRAM	MCDRAM	MCDRAM	MCDRAM
3	MCDRAM	MCDRAM	MCDRAM	MCDRAM
4	DDR4	MCDRAM	MCDRAM	MCDRAM
5	DDR4	MCDRAM	DDR4	MCDRAM
6	DDR4	MCDRAM	DDR4	MCDRAM

unknowns, element-local, e.g. 1x 56x9

unknowns, element-local, e.g. 1x 56x9

star, flux solver, element-local, 11x 9x9

Stiffness and Flux matrices, flux solver, global, e.g. 58x 56x56

→ **At high-order, ~70% of the data can be stored in DDR4 using an out-of-core scheme**

LIBXSMM: Implementation

Interface (C/C++ and FORTRAN API)

Simplified interface for matrix-matrix multiplications

- $c_{m \times n} = c_{m \times n} + a_{m \times k} * b_{k \times n}$ (also full xGEMM)

Highly optimized assembly code generation (inline, *.s, JIT byte-code)

- SSE3, AVX, AVX2, IMCI (KNCni), and AVX-512
- AVX-512 code quality
  - Maximizes number of immediate operands
  - Limits instruction width to 16 Byte/cycle

High level code optimizations

- Implicitly aligned leading dimension (LDC) – allows aligned store instr.
- Aligned load instructions
- Sophisticated data prefetch

License

- Open Source Software (BSD 3-clause license)*, https://github.com/hfp/libxsmm

LIBXSMM AVX512 code for SeisSol (N=9)

```
vmovapd 1792(%rdi), %zmm4
vmovapd 2240(%rdi), %zmm5
vfmadd231pd 16(%rsi){1to8}, %zmm2, %zmm23
vfmadd231pd 16(%rsi,%r15,1){1to8}, %zmm2, %zmm24
vfmadd231pd 16(%rsi,%r15,2){1to8}, %zmm2, %zmm25
vfmadd231pd 16(%rax){1to8}, %zmm2, %zmm26
vfmadd231pd 16(%rsi,%r15,4){1to8}, %zmm2, %zmm27
vfmadd231pd 16(%rax,%r15,2){1to8}, %zmm2, %zmm28
vfmadd231pd 16(%rbx){1to8}, %zmm2, %zmm29
vfmadd231pd 16(%rax,%r15,4){1to8}, %zmm2, %zmm30
vfmadd231pd 16(%rsi,%r15,8){1to8}, %zmm2, %zmm31
vmovapd 2688(%rdi), %zmm6
vmovapd 3136(%rdi), %zmm7
vfmadd231pd 24(%rsi){1to8}, %zmm3, %zmm14
vfmadd231pd 24(%rsi,%r15,1){1to8}, %zmm3, %zmm15
vfmadd231pd 24(%rsi,%r15,2){1to8}, %zmm3, %zmm16
vfmadd231pd 24(%rax){1to8}, %zmm3, %zmm17
vfmadd231pd 24(%rsi,%r15,4){1to8}, %zmm3, %zmm18
vfmadd231pd 24(%rax,%r15,2){1to8}, %zmm3, %zmm19
vfmadd231pd 24(%rbx){1to8}, %zmm3, %zmm20
vfmadd231pd 24(%rax,%r15,4){1to8}, %zmm3, %zmm21
vfmadd231pd 24(%rsi,%r15,8){1to8}, %zmm3, %zmm22
vmovapd 3584(%rdi), %zmm0
```

⟶ **Max. theoretical efficiency: 90%!**

- column-major storage
- Working on all 9 columns and 8 rows simultaneously
- Loads to A (vmovapd) are spaced out to cover L1$ misses
- K-loop is fully unrolled
- B-elements are broadcasted within the FMA instruction to save execution slots
- SIB addressing mode to keep instruction size <= 8 byte for 2 decodes per cycle (16 byte I-fetch per cycle)
- Multiple accumulators (zmm31-xmm23 and zmm22-zmm14) for hiding FMA latencies

Performance Benchmarking Systems

KNL: one Intel® Xeon Phi™ 7520 processor with 68 cores, 1.2 GHz AVX-base core-clock and 1.5 GHz all core Turbo frequency, 1.7 GHz mesh-clock, 16 GB MCDRAM@7.2 GT, 96\,GB DDR4-2400, FLAT/(CACHE or QUADRANT), one core reserved for OS

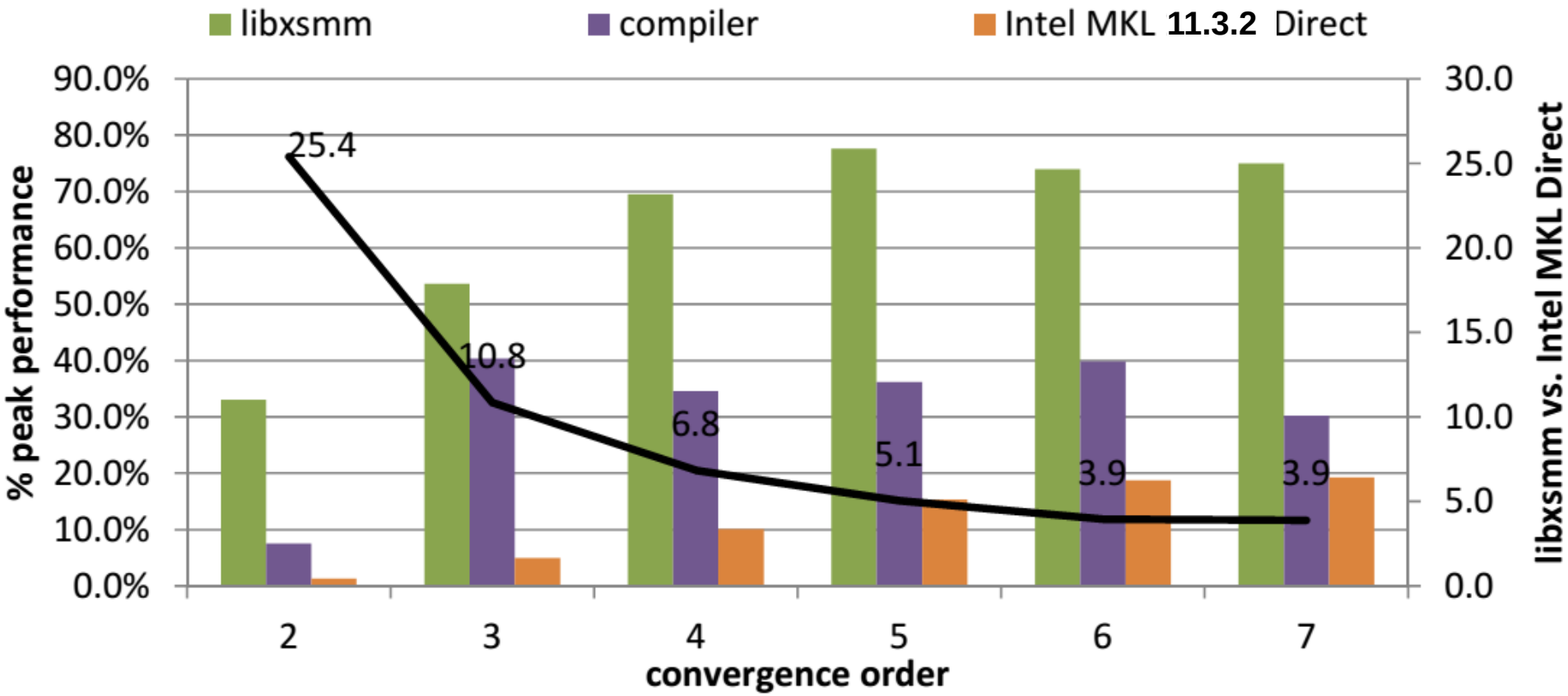
HSX: one Intel® Xeon® E5-2699v3 processor with 18 cores, 1.9 GHz at AVX-base frequency and up to 2.6 GHz Turbo frequency, 64 GB of DDR4-2133

KNC: one Intel® Xeon Phi™ 7120A coprocessor in native mode with 61 cores, 1.24 GHz base and 1.33 GHz Turbo frequency, 16 GB of GDDR5, one core reserved for OS

→We carry out single socket comparisons as Intel's reference platforms offer the same amount of KNL and HSX/BDX(Intel® Xeon® E5v4) sockets per rack-U!

Kernel Performance per Convergence Order

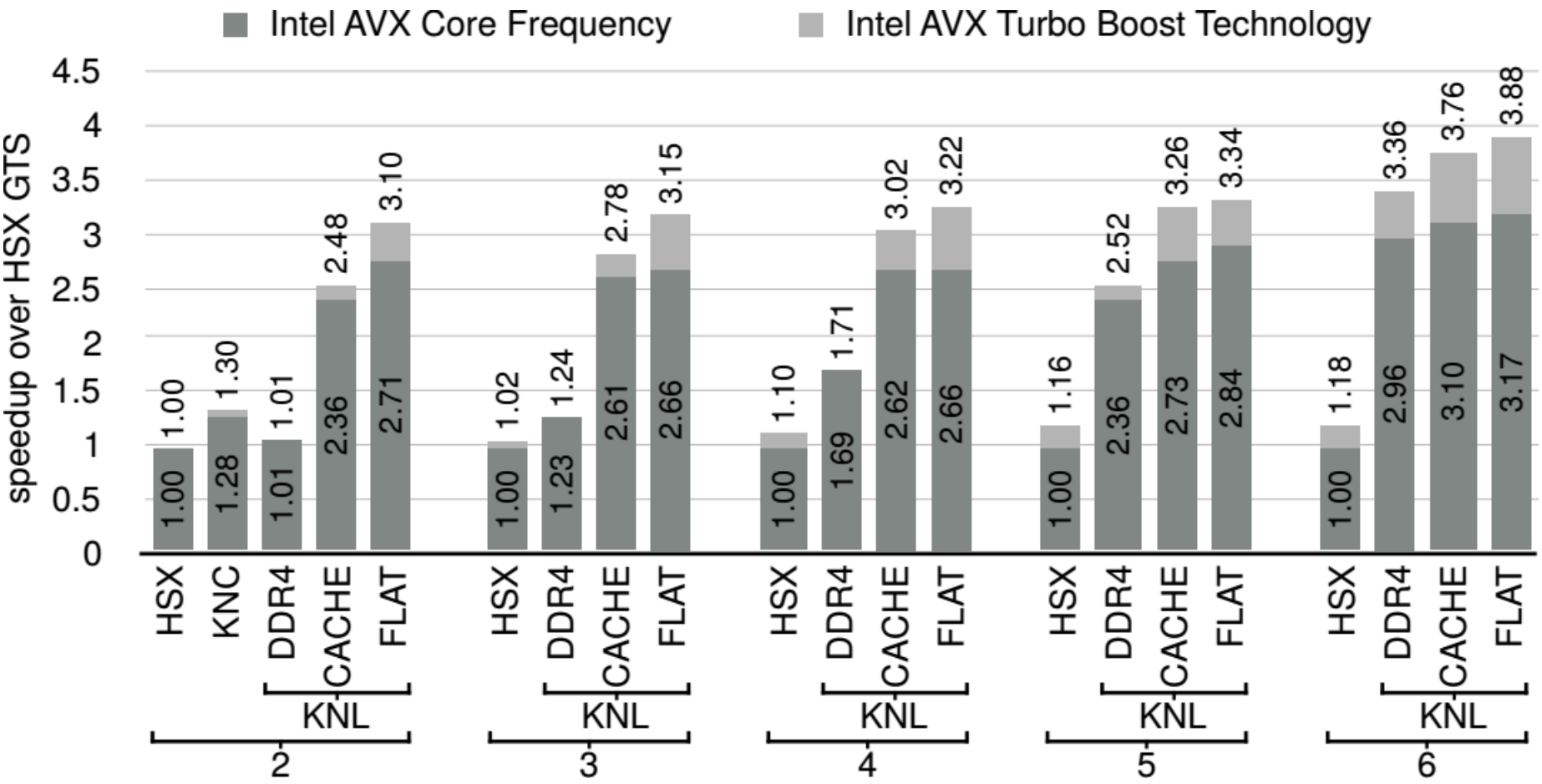
for selected, most important left-side operator, single core



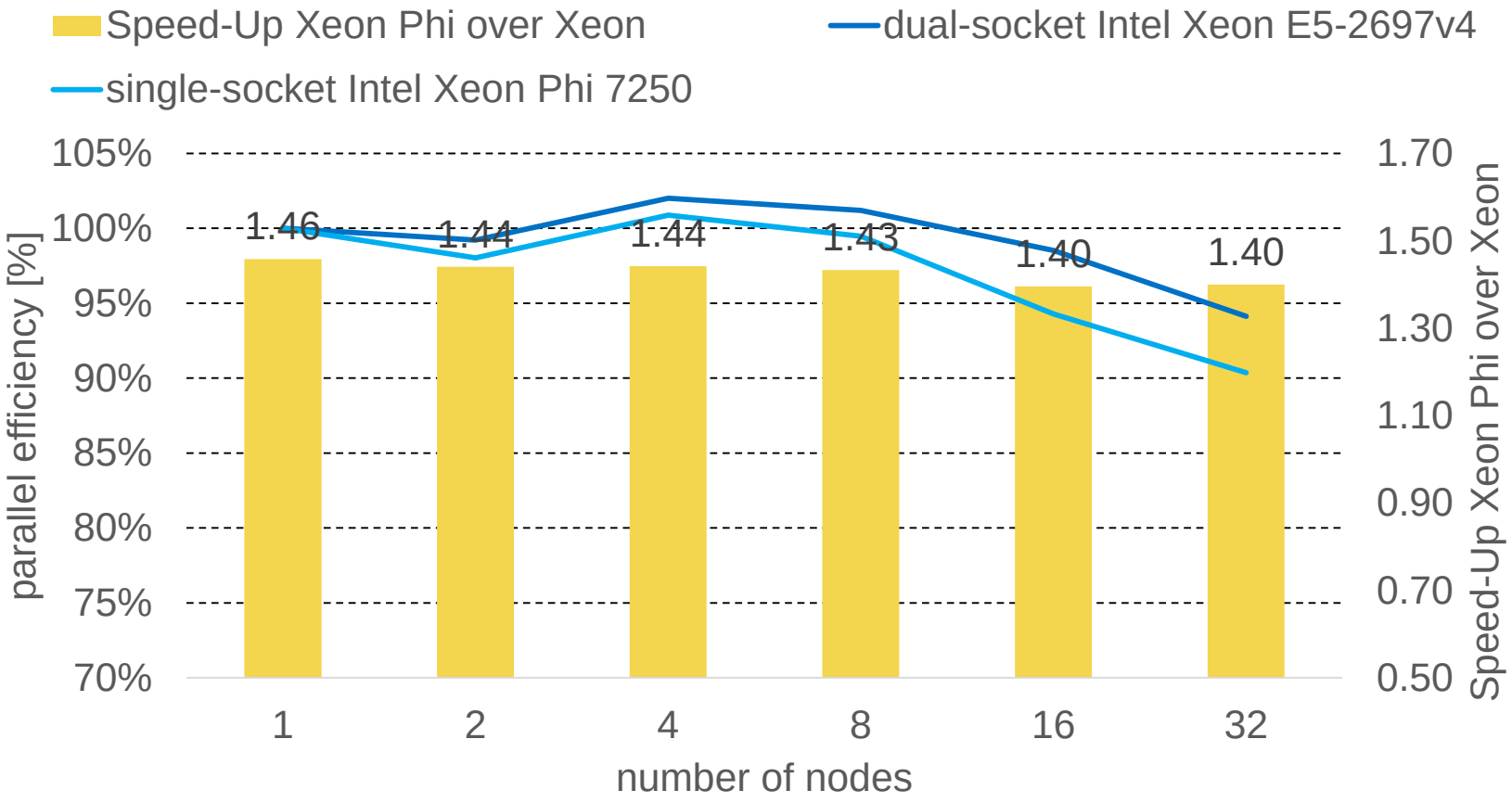
$B_{\mathcal{O}} \times 9 \times B_{\mathcal{O}}$ shapes $B_2 = 4$, $B_3 = 10$, $B_4 = 20$, $B_5 = 35$, $B_6 = 56$ and $B_7 = 84$

Not accounting for loop management, LD/ST to C matrix max. performance is 90% peak!
N=9 -> 1 LD + 9 FMAs is the basic block using all tricks gives 2 LD + 18 FMAs
-> 10 cycles out of which 9 are compute -> 90% peak.

Mount Merapi (Volcano on Java) - Performance



Cluster Strong Scaling of the Mount Merapi Setup



- On 32 nodes, only 47K elements per nodes, <1K per core!
- Measured on Intel's Endeavor Cluster. Each node is equipped with one Intel Omni-Path HFI PCIe x16 card offering 100 Gbps bandwidth.

NekBox and LIBXSMM

NekBox

NekBox is a stripped-down of Nek5000 for box-shaped domains

NekBox solves the incompressible Navier-Stokes equations:

$$\frac{\partial u}{\partial t} + u \cdot \nabla u = -\frac{1}{\rho}\nabla p + \nu \nabla^2 u + f \qquad \nabla \cdot u = 0$$

Incl. advection-diffusion equations for scalar variables such as temperature or mass fractions.

Nek uses the spectral element method (SEM):

- A tensor product of Gauss-Lobatto-Legendre (GLL) quadrature points within each element -> N^3 DOFs per element
- Continuity across elements -> forming a mesh (using direct stiffness summation)

Operators are written as element local operators:

$$A = (A_x \times I_y \times I_z) + (I_x \times A_y \times I_z) + (I_x \times I_y \times A_z)$$

which reduces the complexity from O(N^6) to O(N^4).

NekBox's main compute routines

A typical NekBox run spends <1% in sparse computations & communications, ~40% in vector-vector or matrix-vector operations**, ~60%** matrix-matrix operations.

Helmholtz operator:

```
Hu(:,:,:) = gx(:,:,:) * matmul(Kx(:,:), reshape(u, (/N, N*N/)))
do i = 1, n
  Hu(:,:,i) += gy(:,:,i) * matmul(u(:,:,i), KyT(:,:))
enddo
Hu(:,:,:) += gz(:,:,:) * matmul(reshape(u, (/N*N, N/)), KzT(:,:))
Hu(:,:,:) = h1 * Hu(:,:,:) + h2 * M(:,:,:) * u(:,:,:)
```

Basis transformation:

```
tmp_x = matmul(Ax, u)
do i = 1, n
  tmp_y(:,:,i) = matmul(tmp_x(:,:,i), AyT)
enddo
v = matmul(tmp_y, AzT)
```

Gradient calculation:

```
dudx = matmul(Dx, u)
do i = 1, n
  dudy(:,:,i) = matmul(u(:,:,i), DyT)
enddo
dudz = matmul(u, DzT)
```

→ **Batched GEMM is not beneficial as it would mean loosing locality**

Element Updates with non-temporal Stores

- NekBox performs vector stores that overwrite memory on necessarily unaligned data
- Regular unaligned stores would issue RFOs (read for ownership), which consume read BW
- Aligned non-temporal stores via peeling allow us to override the data without RFOs, increasing “useful” bandwidth

```
void stream_vector_copy( const double* i_a,
                         double*       io_c,
                         const int     i_length) {
  int l_n = 0;
  int l_trip_prolog = 0;
  int l_trip_stream = 0;

  /* init the trip counts to determine aligned middle section */
  stream_init( i_length, (size_t)io_c, &l_trip_prolog, &l_trip_stream );

  /* run the prologue */
  for ( ; l_n < l_trip_prolog;  l_n++ ) {
    io_c[l_n] = i_a[l_n];
  }
  /* run the bulk, using streaming stores */
  for ( ; l_n < l_trip_stream;  l_n+=8 ) {
    _mm256_stream_pd( &(io_c[l_n]),   _mm256_loadu_pd(&(i_a[l_n]))   );
    _mm256_stream_pd( &(io_c[l_n+4]), _mm256_loadu_pd(&(i_a[l_n+4])) );
  }
  /* run the epilogue */
  for ( ; l_n < i_length;  l_n++ ) {
    io_c[l_n] = i_a[l_n];
  }
}
```

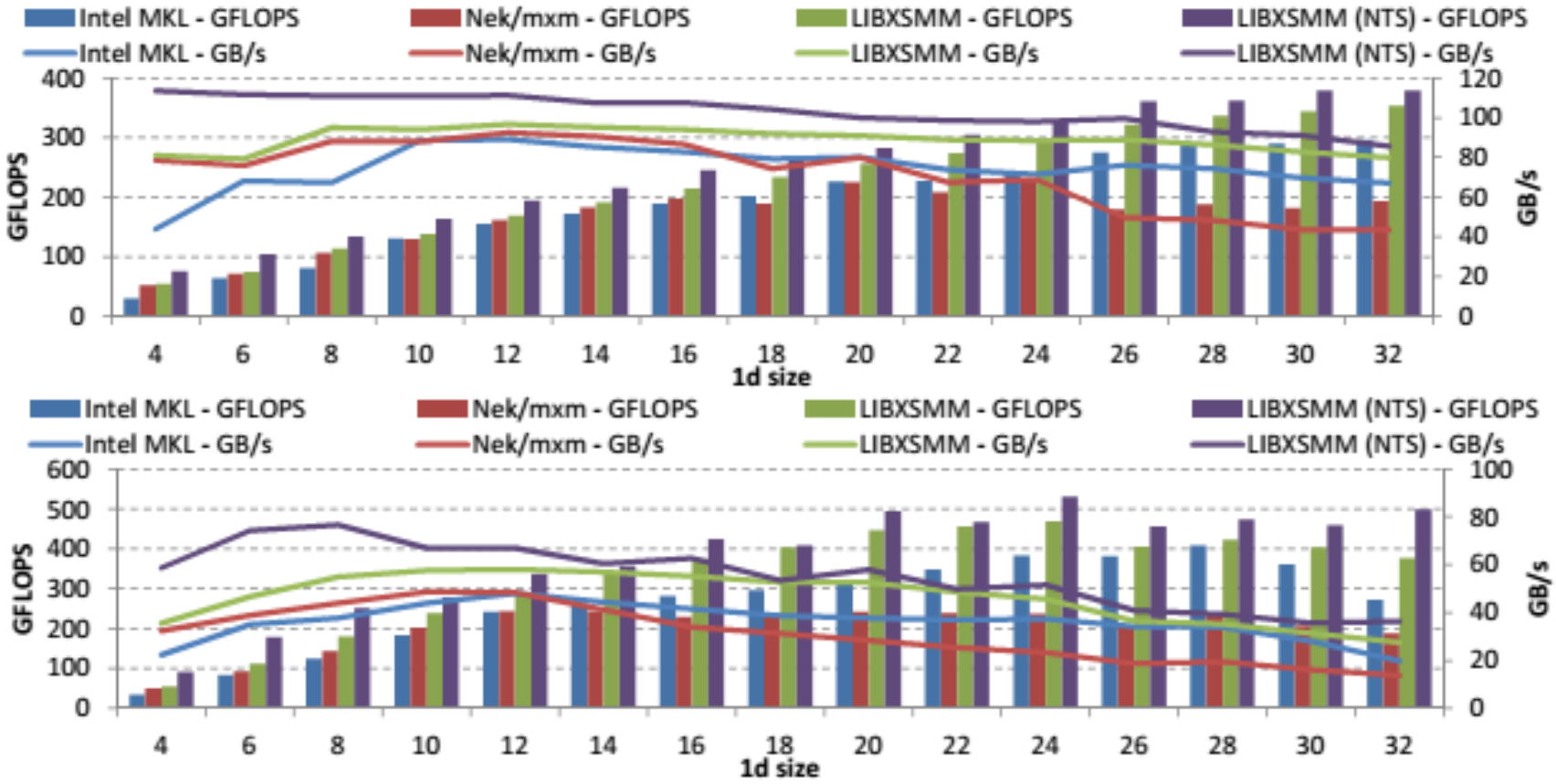
Systems we used in this Study

- Mira is a IBM BlueGene/Q with 49,152 nodes hosted at ANL in the US. Each node has 16 cores with 4 hardware threads per core and can support 204.8 GFLOPS and 30 GiB/s main memory bandwidth. (FLOP/BYTES ~ 6)

- Shaheen is a Cray XC40 with 6144 nodes hosted at KAUST in Saudi Arabia. Each node has two Intel® Xeon® E5-2698v3 (code-named Haswell) processors with 16 cores each and can support around 1177.6 GFLOPS and 101.6 GiB/s main memory bandwidth, combined on both NUMA domains. (FLOPS/BYTE ~10)

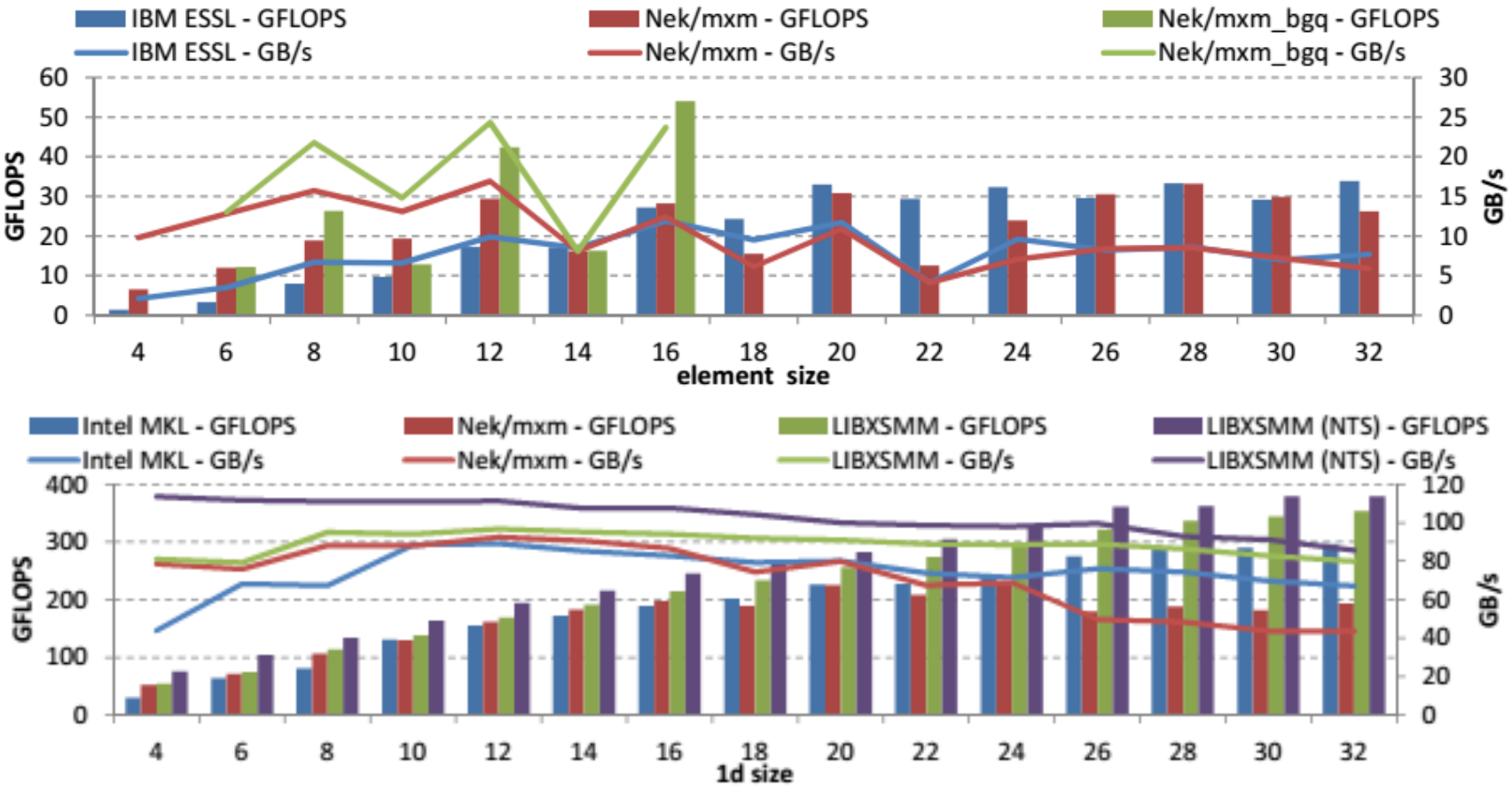
→ Shaheen's cores therefore have 2.9× the floating point and 1.7× the memory bandwidth of Mira's BlueGene/Q cores.

Helmholtz Op. and Basis Transf. on Shaheen



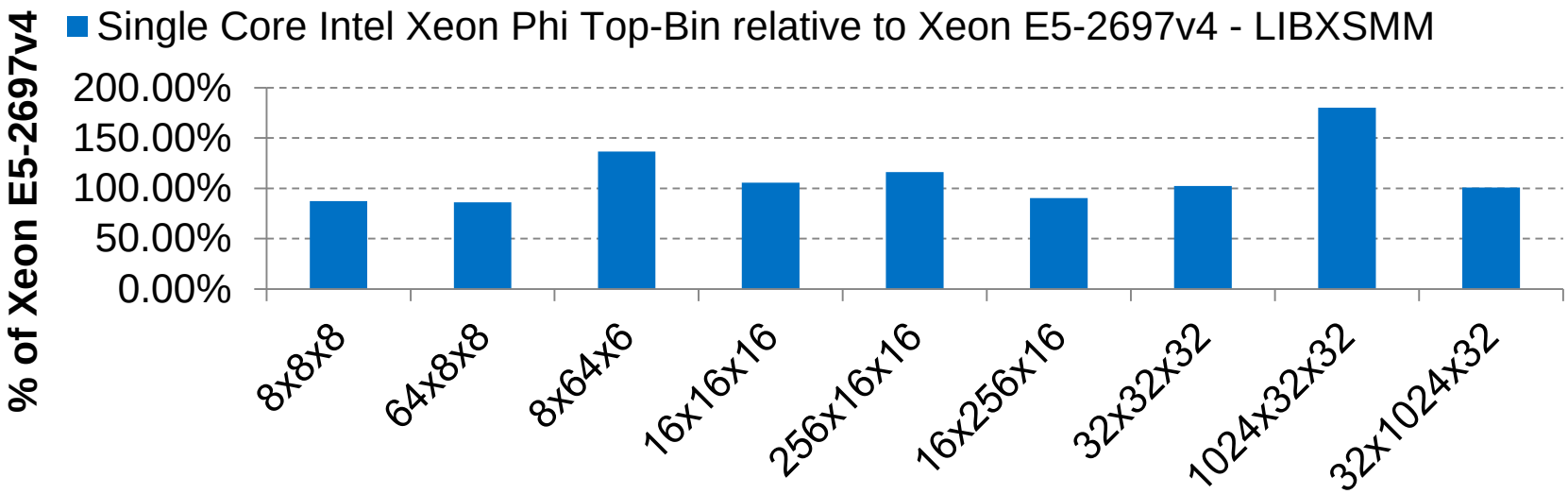
Performance of reproducers for the Helmholtz operator (top) and Basis Transformation (bottom) using different implementation for the small matrix multiplications. NTS denotes the usage of non-temporal stores. Measured on Shaheen (32 cores of HSW-EP, 2.3 GHz)

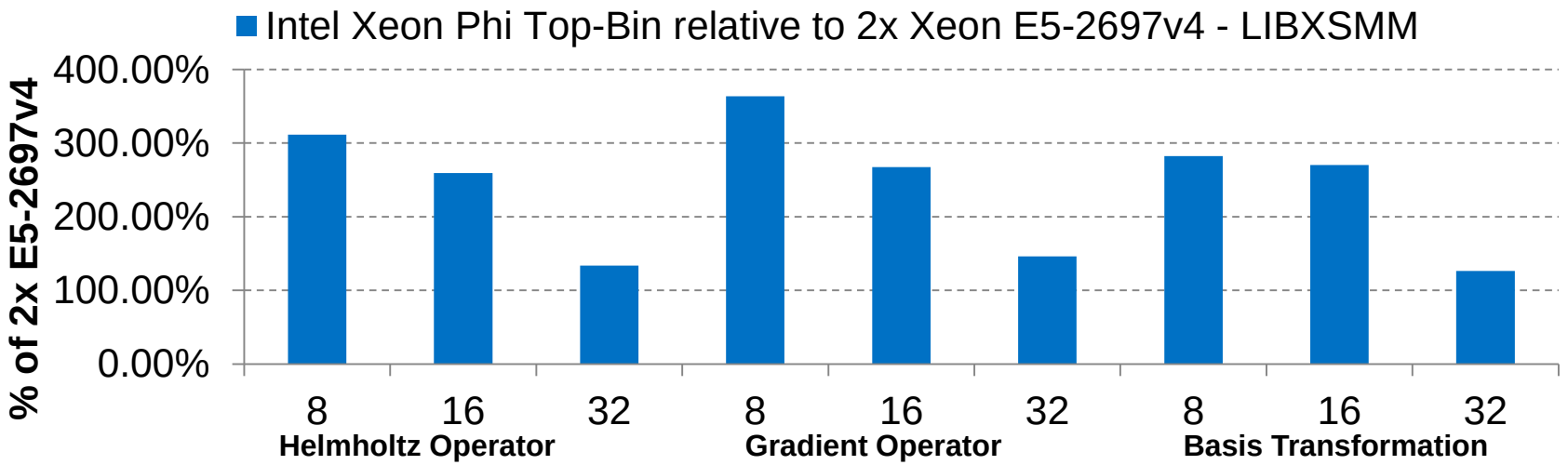
Helmholtz-Operator on Mira (BG/Q) vs. Shaheen



Performance of reproducer for the Helmholtz operator reproducer on Mira (top) and Shaheen (bottom). IBM Blue Gene/Q doesn't offer unaligned memory support for vector instructions and Intel Xeon benefits from more flexible hardware and its optimal utilization through LIBXSMM. mxm_bgq is an assembly library for Mira.

Xeon Phi 7250 (Knights Landing) results





Legal Disclaimer & Optimization Notice

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804


intel®
experience
what's inside™